A Billion SMT Queries a Day (Invited Paper)

Neha Rungta(✉)

Amazon Web Services, Seattle, USA
rungta@amazon.com

Abstract. Amazon Web Services (AWS) is a cloud computing services provider that has made significant investments in applying formal methods to proving correctness of its internal systems and providing assurance of correctness to their end-users. In this paper, we focus on how we built abstractions and eliminated specifications to scale a verification engine for AWS access policies, Zelkova, to be usable by all AWS users. We present milestones from our journey from a thousand SMT invocations daily to an unprecedented billion SMT calls in a span of five years. In this paper, we talk about how the cloud is enabling application of formal methods, key insights into what made this scale of a billion SMT queries daily possible, and present some open scientific challenges for the formal methods community.

Keywords: Cloud Computing · Formal Verification · SMT Solving

1 Introduction

Amazon Web Services (AWS) has made significant investments in developing and applying formal tools and techniques to prove the correctness of critical internal systems and provide services to AWS users to prove correctness of their own systems [24]. We use and apply a varied set of automated reasoning techniques at AWS. For example, we use (i) bounded model checking [35] to verify memory safety properties of boot code running in AWS data centers and of real-time operating system used in IoT devices [22,25,26], (ii) proof assistants such as EasyCrypt [12] and domain-specific languages such as Cryptol [38] to verify cryptographic protocols [3,4,23], (iii) HOL-Lite [33] to verify the BigNum implementation [2], (iv) P [28] to test key storage components in Amazon S3 [18], and (v) Dafny [37] to verify key authorization and crypto libraries [1]. Automated reasoning capabilities for external AWS users leverage (i) data-flow analysis [17] to prove correct usage of cloud APIs [29,40], (ii) monotonic SAT theories [14] to check properties of network configurations [5,13], and (iii) theories for strings and automaton in SMT solvers [16,39,46] to provide security for access controls [6,19].

This paper describes key milestones in our journey of generating billion SMT queries a day in the context of AWS Identity and Access Management (IAM). IAM is a system for controlling access to resources such as applications, data, and workload in AWS. Resource owners can configure access by writing *policies* that describe when to allow and deny user requests that access the resource. These configurations are expressed in the IAM policy language. For example, Amazon Simple Storage Service (S3) is an object storage service that offers data durability, availability, security, and performance. S3 is used widely to store and protect data for a range of applications. A *bucket* is a fundamental container in S3 where users can upload unlimited amounts of data in the form of objects. Amazon S3 supports fine-grained access control to the data based on the needs of the user. Ensuring that only intended users have access to their resource is important for the security of the resource. While the policy language allows for compact specifications of expressive policies, reasoning about the interaction between the semantics of different policy statements can be challenging to manually evaluate, especially in large policies with multiple operators and conditions.

To help AWS users secure their resources, we built Zelkova, a policy analysis tool designed to reason about the semantics of AWS access control policies. Zelkova translates policies and properties into Satisfiability Modulo Theories (SMT) formulas and uses SMT solvers to prove a variety of security properties such as "*Does the policy grant broad public access?*" [6]. The SMT encoding uses the theory of strings, regular expressions, bit vectors, and integer comparisons. The use of the wildcards $*$ (any number of characters) and ? (exactly one character) in the string constraints makes the decision problem PSPACE-complete. Zelkova uses a portfolio solver, where it invokes multiple solvers in the backend and uses the results from the solver that returns first, in a winner takes all strategy. This allows us to leverage the diversity among solvers and quickly solve queries—a couple hundred milliseconds to tens of seconds. A sample of AWS services that integrate Zelkova includes Amazon S3 (object storage), AWS Config (change-based resource auditor), Amazon Macie (security service), AWS Trusted Advisor (compliance to AWS best practices), and Amazon GuardDuty (intelligent threat detection). Zelkova drives preventative control features such as Amazon S3 Block Public Access and visibility into who outside an account has access to its resources [19].

Zelkova is an automated reasoning tool developed by formal methods experts and requires some degree of expertise in formal methods to use it. We cannot expect all AWS users to be experts in formal methods, have the time to be trained in the use of formal methods tools, or even be experts in the cloud domain. In this paper, we present the three pillars of our solution that enable Zelkova to be used by *all AWS users*. Using a combination of techniques such as eliminating specifications, domain-specific abstractions, and advances in SMT solvers we make the power of Zelkova available to all AWS users.

2 Eliminate Writing Specifications

End users will not write a specification

Zelkova follows a traditional verification approach where it takes as input a policy and a specification, and produces a yes or no answer. We have developers and cloud administrators who author policies to govern access to cloud

```
- Effect: Allow
  Condition:
    StringEquals:
      SrcVpc:
        - vpc-a
        - vpc-b
- Effect: Allow
  Condition:
    StringEquals:
      OrgID: o-2
- Effect: Deny
  Condition:
    StringEquals:
      SrcVpc: vpc-b
    StringNotEquals:
      OrgID: o-1
```

Fig. 1. An example AWS policy

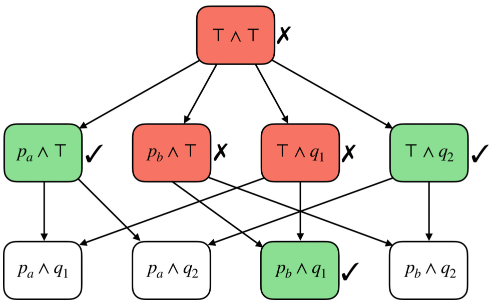


Fig. 2. Stratified abstraction search tree

resources. We have someone else, a security engineer, who writes a specification of what is considered acceptable. The automated reasoning engine ZELKOVA does the verification and returns a yes or no answer. This approach is effective for a limited number of use cases, but it is hard to scale to all AWS users. The bottleneck to scaling the verification effort is the *human effort required to specify what is acceptable behavior*. The SLAM work had similar a observation about specifications; for use of Static Driver Verifier, they needed to provide the tool as well as the specification [7]. A person has to put in a lot of work upfront to define acceptable behavior and only at the end of the process, they get back an answer—a boolean. It's a single bit of information for all the work they've put in. They have no information about whether they had the right specification or whether they wrote the specification correctly.

To scale our approach to all AWS users, we had to fundamentally rethink our approach and completely remove the bottleneck of having people write a specification. To achieve that, we flipped the rules of the game and made the automated reasoning engine responsible for specification. We had the machine put in the upfront cost. Now it takes as input a policy and returns a detailed set of findings (declarative statements about what is true of the system). These findings are presented to a user, the security engineer, who reviews these findings and makes decisions about whether these findings represent valid risks in the system that should be fixed or are acceptable behaviors of the system. Users are now taking the output of the machine and saying "yes" or "no".

2.1 Generating Possible Specifications (Findings)

To remove the bottleneck of specification, we changed the question from is this policy correct? to who has access?. The response to the former is a boolean while the response to the latter is a set of findings. AWS access control policies specify *who* has access to a given resource, via a set of `Allow` and `Deny` statements that grant and prohibit access, respectively. Figure 1 shows a simplified policy specifying access to an AWS resource. This policy specifies conditions on the cloud-based network (known as a VPC) for which the request originated and on the organizational Amazon customer (referred to by an Org ID) who made the request. The first statement *allows* access to any request whose `SrcVpc` is either `vpc-a` *or* `vpc-b`. The second statement *allows* access to any request whose `OrgId` is `o-2`. However, the third statement *denies* access from `vpc-b` *unless* the `OrgId` is `o-1`.

For each request, access is granted only if: (a) *some* `Allow` statement matches the request, and (b) *none* of the `Deny` statements match the request. Consequently, it can be quite tricky to determine what accesses are allowed by a given policy. First, individual statements can use regular expressions, negation, and conditionals. Second, to know the effect of an allow statement, one must consider all possible deny statements that can *overlap* with it, *i.e.*, can refer to the same request as the allow. Thus, policy verification is not *compositional*, in that we cannot determine if a policy is "correct" simply by *locally* checking that each statement is "correct." Instead, we require a *global* verification mechanism, that simultaneously considers all the statements and their subtle interactions, to determine if a policy grants only the intended access.

For the example policy sketch shown in Fig. 1, access can be summarized through a set of three findings, which say that access is granted to a request iff:

– Its `SrcVpc` is `vpc-a`, *or*,
– Its `OrgId` is `o-2`, or,
– Its `SrcVpc` is `vpc-b` *and* its `OrgId` is `o-1`.

The findings are sound as no other requests are granted access. The findings are mostly precise; most of the requests match the conditions that are granted access. The finding "`OrgId` is `o-2`" also includes some requests that are not allowed, *e.g.*, when `SrcVpc` is `vpc-b`. To help understandability of the findings, we sacrifice this precision. Precise findings would need to include negation, and that would add complexity for the users to make decisions. Finally, the findings compactly summarize the policy in three positive statements declaring *who* has access. In principle, the notion of compact findings is similar to abstract counterexamples or minimizing counterexamples [21,30,32]. Since the findings are produced by the machine and already verified to be true, we have a person deciding if they *should be* true. The human is making a judgment call and expressing intent.

We use stratified predicate abstraction for computing the findings. Enumerating all possible requests is computationally *intractable*, and even if it were not, the resulting set of findings is far too large and hence *useless*. We tackle the problem of summarizing the super-astronomical request-space by using *predicate abstraction*. Specifically, we make a syntactic pass over the policy to extract the set of constants that are used to constrain access, and we use those constants to generate a family of predicates whose conjunctions compactly describe partitions of the space of all requests. For example, from the policy in Fig. 1 we would extract the following predicates

$$\begin{array}{lll} p_a \doteq \mathsf{SrcVpc} = \mathsf{vpc\text{-}a}, & p_b \doteq \mathsf{SrcVpc} = \mathsf{vpc\text{-}b}, & p_\star \doteq \mathsf{SrcVpc} = \star, \\ q_1 \doteq \mathsf{OrgId} = \mathsf{o\text{-}1}, & q_2 \doteq \mathsf{OrgId} = \mathsf{o\text{-}2}, & q_\star \doteq \mathsf{OrgId} = \star. \end{array}$$

The first row has three predicates describing the possible value of the `SrcVpc` of the request: that it equals `vpc-a` or `vpc-b` or some value other than `vpc-a` and `vpc-b`.

$\boxed{p_a \wedge q_1}$ ✓	$\boxed{p_a \wedge q_2}$ ✓	$\boxed{p_a \wedge q_\star}$ ✓
$\boxed{p_b \wedge q_1}$ ✓	$\boxed{p_b \wedge q_2}$ ✗	$\boxed{p_b \wedge q_\star}$ ✗
$\boxed{p_\star \wedge q_1}$ ✗	$\boxed{p_\star \wedge q_2}$ ✓	$\boxed{p_\star \wedge q_\star}$ ✗

Fig. 3. Cubes generated by the predicates $p_a, p_b, p_\star, q_1, q_2, q_\star$ generated from the policy in Fig. 1 and the result of querying Zelkova to check if the the requests corresponding to each cube are granted access by the policy.

Similarly, the second row has three predicates describing the value of the `OrgId` of the request: that it equals `o-1` or `o-2` or some value other than `o-1` and `o-2`.

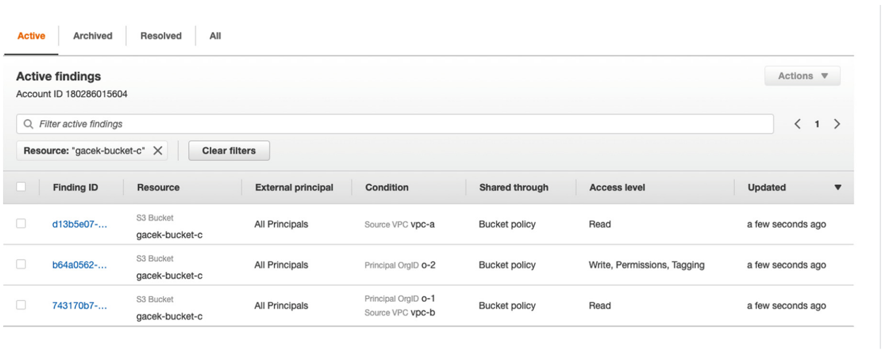
We can compute findings by enumerating all the *cubes* generated by the above predicates and querying Zelkova to determine if the policy allows access to the requests described by the cube. The enumeration of cubes is common in SAT solvers and other predicate abstraction based approaches [8,15,36]. The set of all the cubes are shown in Fig. 3. The chief difficulty with enumerating all the cubes *greedily* is that we end up eagerly *splitting-cases* on the values of fields when that may not be required. For example, in Fig. 3, we split cases on the possible value of `OrgId` even though it is irrelevant when `SrcVpc` is `vpc-a`. This observation points the way to a new algorithm where we *lazily* generate the cubes as follows. Our algorithm maintains a *worklist* of minimally refined cubes. At each step, we (1) ask Zelkova if the cube allows an access that is not covered by any of its refinements; (2) if so, we add it to the set of findings; and (3) if not, we refine the cube "point-wise" along the values of each field individually and add the results to the worklist. The above process is illustrated in Fig. 2.

The specifications or findings generated by the machine are presented in the context of the access control domain. The developers do not have to learn a new means to specify correctness, think about what they want to be correct of the system, or check the completeness of their specifications. This is a very important lesson that we need to apply across many other applications for formal methods to be successful at scale. The challenge here is the specifics depend on the domain.

3 Domain-Specific Abstractions

It's all about the end user

Zelkova was developed by formal methods subject matter experts who learnt domain of AWS access control policies. Once we had the analysis engine, we faced the same challenges all other formal methods tool developers had before us. How do we make it accessible to all users? One hard earned lesson was "eliminating the need for specifications" as discussed in the previous section. But that was only part of the answer. There was a lot more to do. Many more questions to answer—How do we get users to use it? How do we present the results to the



Active findings
Account ID 180286015604

Resource: "gacek-bucket-c"

	Finding ID	Resource	External principal	Condition	Shared through	Access level	Updated
☐	d13b5e07-...	S3 Bucket gacek-bucket-c	All Principals	Source VPC vpc-a	Bucket policy	Read	a few seconds ago
☐	b64a0562-...	S3 Bucket gacek-bucket-c	All Principals	Principal OrgID o-2	Bucket policy	Write, Permissions, Tagging	a few seconds ago
☐	743170b7-...	S3 Bucket gacek-bucket-c	All Principals	Principal OrgID o-1 Source VPC vpc-b	Bucket policy	Read	a few seconds ago

Fig. 4. Interface that presents Access Analyzer findings to users.

users? How do the results stay updated? The answer was to design and build domain-specific abstractions. Do one thing and do it really well.

We created a higher level service on top of Zelkova called IAM Access Analyzer. We provide a one-click way to enable Access Analyzer for an AWS account or AWS Organization. An account in AWS is a fundamental construct that serves as a container for the user's resources, workloads, and data. Users can create policies to grant access to resources in their account to other users. In Access Analyzer, we use the account as a *zone of trust*. This abstraction lets us say that access to resources by users within their zone of trust is considered safe. But access to resources outside their zone of trust is potentially unsafe.

Once a user enables Access Analyzer, we use stratified predicate abstraction to analyze the policies and generate findings showing which users outside the zone of trust have access to resources. We had to shift from a mode where Zelkova can answer "any access query" to Zelkova can enumerate "who has access to what". This brings to attention the permissions that could lead to unintended access of data. While this idea seems simple in hindsight, it took us a couple of years to figure out the right abstraction for the domain. It can be used by all AWS users. They did not need to be experts in the area of formal methods or even have deep understanding of how access control in the cloud worked.

Each finding includes details about the resource, the external entity with access to it, and the permissions granted so that the user can take appropriate action. We present example findings in Fig. 4. Note these findings are not presented as SMT-lib formulas but rather in the domain that the user expects—AWS access control constructs. These map to the findings presented in the previous section for Fig. 1. Users can view the details included in the finding to determine whether the access is intentional or a potential risk that the user should resolve.

Most automated reasoning tools are run as a one-off: prove something, and then move on to the next challenge. In the cloud environment this was not the case. Doing the analysis once was not sufficient in our domain. We had to design a means to continuously monitor the environment and changes to access control policies within the zone of trust and update the findings based on that. To that end, Access Analyzer analyzes these policies if a user adds a new policy, or changes an existing policy, and either generates new findings, or removes findings, or updates the existing findings. Access Analyzer also analyzes all policies periodically, to ensure that in a rare case, if a change event to the policy is missed by the system, it is still able to keep the findings updated. The ease of enablement, just-in-time analysis on updates, and periodic analysis across all policies are the key factors in getting us to a billion queries daily.

4 SMT Solving at Cloud Scale

Every query matters

The use of SMT solving in AWS features and services means that millions of users are relying on the correctness and timeliness of the underlying solvers for the security of their cloud infrastructure. The challenges around correctness and timeliness in solver queries have been well studied in the automated reasoning community, but they have been treated as independent features. Today, we are generating a billion SMT queries every day to support various use cases across a wide variety of AWS services. We have discovered an intricate dependency between correctness and timeliness that manifests at this scale.

4.1 Monotonicity in Runtimes Across Solver Versions

Zelkova uses a portfolio solver to discharge its queries. When given a query, Zelkova invokes multiple solvers in the backend and uses the results from the solver that returns first, in a winner takes all strategy [6]. The portfolio approach allows us to leverage the diversity amongst solvers. One of our goals is to leverage the latest advancements in the SMT solver community. SMT solver researchers and developers are fixing issues, making improvements to existing features, adding new theories, adding features such as generation of proofs, and making other performance improvements. Before deploying a new version of the solver within the production environment, we perform extensive offline testing and benchmarking to gain confidence in the correctness of the answers, performance of the queries, and ensure there are no regressions.

While striving for correctness and timeliness, one of the challenges we face is that new solver versions are not monotonically better in their performance than their previous version. A solution that works well in the cloud setting is a massive portfolio, sometimes even containing older versions of the same solver. This presents two issues. One, when we discover a bug in an older version of the solver, we need to patch this old version. This creates an operational burden of maintaining many different versions of the different solvers. Two, when the number of solvers increases, we need to ensure that each solver provides a correct result. Checking the correctness of queries that result in SAT is straightforward, but SMT solvers need to provide proof for the UNSAT queries. The proof generation and checking needs to be timely as well.

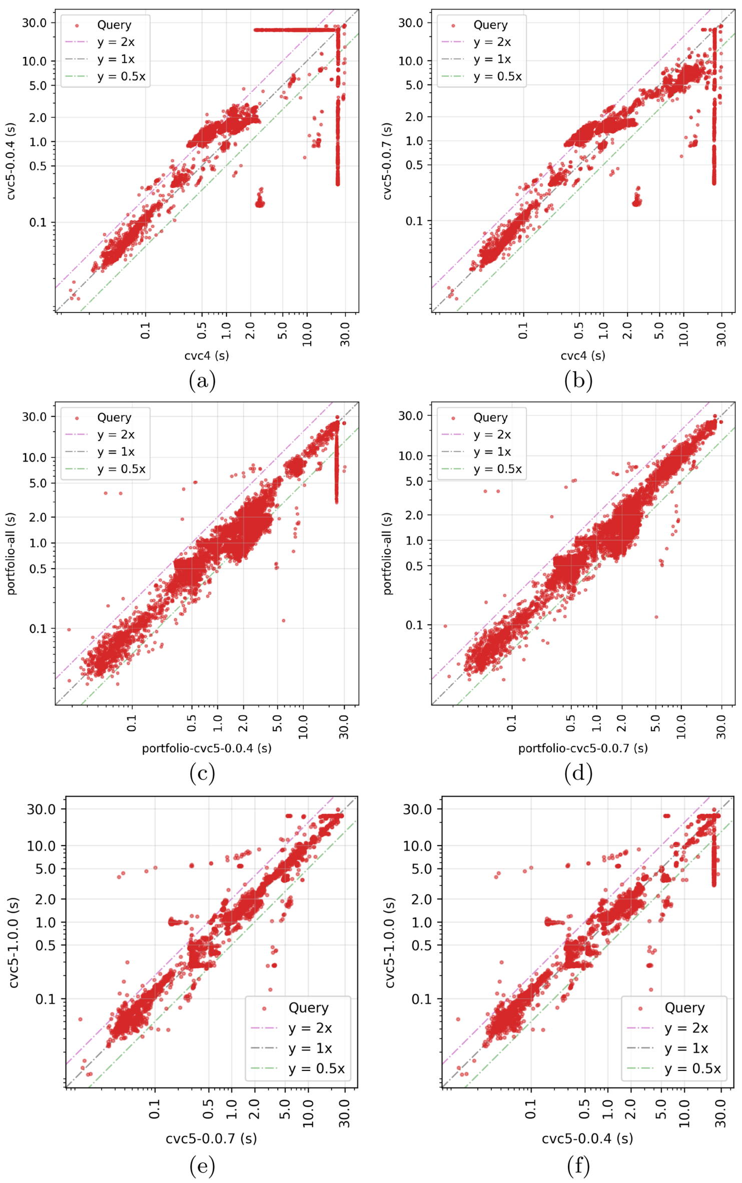


Fig. 5. Comparing the runtime for solving SMT queries generated by ZELKOVA by CVC4 and the different cvc5 versions (a) CVC4 vs. cvc5 version 0.0.4, (b) CVC4 vs. cvc5 version 0.0.7. Comparing the runtimes of winner take all in the portfolio solver of ZELKOVA with: (c) a portfolio solver consisting of Z3 sequence string solver, Z3 automata solver, and cvc5 version 0.0.4 (d) a portfolio solver consisting of Z3 sequence string solver, Z3 automata solver, and cvc5 version 0.0.7. Evaluating the performance of the latest cvc5 version 1.0.0 with its older versions (e) cvc5 version 0.0.4 and (f) cvc5 version 0.0.7

In the Zelkova portfolio solver [6], we use CVC4, and our original goal was to replace CVC4 with the then latest version of cvc5 (version 0.0.4)[1]. We wanted to leverage the proof checking capabilities of cvc5 to ensure the correctness of UNSAT queries [11]. To check the timeliness requirements, we ran experiments across our benchmarks, comparing the results of CVC4 to those of cvc5 (version 0.0.4). The results across a representative set of queries are shown in Fig. 5(a). In the graph we have approximately 15,000 SMT queries that are generated by Zelkova; we select a distribution of queries that are solved between 1 s and 30 s, after which the solver process is killed and a timeout is reported. Some queries that are not solved by CVC4 within the time bound of 30 s are now being solved by cvc5 (version 0.0.4), as seen by the points in the graph along the y-axis on the extreme right. However, cvc5 (version 0.0.4) times out on some queries that are solved by CVC4, as seen by the points on the top of the graph.

The results presented in Fig. 5(b) are not surprising given that the problem space is computationally hard, and there is an inherent randomness in search heuristics within SMT solvers. In an evaluation of cvc5, the authors discuss examples where CVC4 outperforms cvc5 [10]. But this poses a challenge for us when we are using the result of these solvers in security controls and services that millions of users rely on. The changes did not meet the timeliness requirement of continuing to solve the queries within 30 s. When a query times out, to be sound, the analysis marks the bucket as public. The impact of a query timing out, that was previously being solved, will lead to the user not being able to access the resource. This is unexpected for the user because there was no change in their configuration.

For example, consider the security checks in the Amazon S3 Block Public Access that block requests based on the results of the analysis. In this context, suppose that there was a bucket marked as “not public” based on the results of a query, and now that same query times out; the bucket will be marked as “public”. This will lock down access to the bucket and the intended users will not be able to access it. Even a single regression that leads to loss of access for the user is not an acceptable change. As another example, these security checks are also used by IoT devices. In the case of a smart lock, a time out in the query that was previously being solved could lead to a loss of access to the user’s home. The criticality of these use cases combined with the end user expectation is a key challenge in our domain.

We debugged and fixed the issue in cvc5 that was causing certain queries to time out. But even with this fix, CVC4 was 2x faster than cvc5 for many easier problems that took 1 s to solve originally. This slowdown was significant for us because Zelkova is called in the request path of security controls such as Amazon S3 Block Public Access. When a user attempts to attach a new access control policy or update an existing one, a synchronous call is made to Zelkova and the corresponding portfolio solvers to determine if the access control policy

[1] Note that while this section talks in detail about the CVC solver, the observations are common across all solvers. We select the results of the CVC solver as a representative because it is a mature solver with an active community.

being attached grants unrestricted public access or not. The bulk of the analysis time is spent in the SMT solvers, so doubling the analysis time for queries can lead to a degraded user experience. Where and how the analysis results are used plays an important role in how we track changes to the timeliness of the solver queries.

Our solution was to *add a new solver to the portfolio rather then replace an existing solver*. We added cvc5 (version 0.0.7) to the existing portfolio of solvers consisting of CVC4, Z3 with the sequence string solver, and a custom Z3-based automata solver. When we started the evaluation of cvc5, we did not plan to add a new version of the CVC solver to the portfolio. We had expected to the latest version of cvc5 to be comparable in timeliness to CVC4. We worked closely with the CVC developers and cvc5 was better on many queries, but it did not meet our timeliness requirements on all queries. This led to our decision to add cvc5 (version 0.0.7) to the Zelkova portfolio solver.

The results of comparing the portfolio solvers of two Z3 solvers, CVC4 and cvc5 (version 0.0.4) with a winner take all and portfolio solver *without* cvc5 (version 0.0.4) is shown in Fig. 5(c). The same configuration now with cvc5 (version 0.0.7) is shown in Fig. 5(d). The results show that the portfolio solving approach that Zelkova takes in the cloud is an effective one.

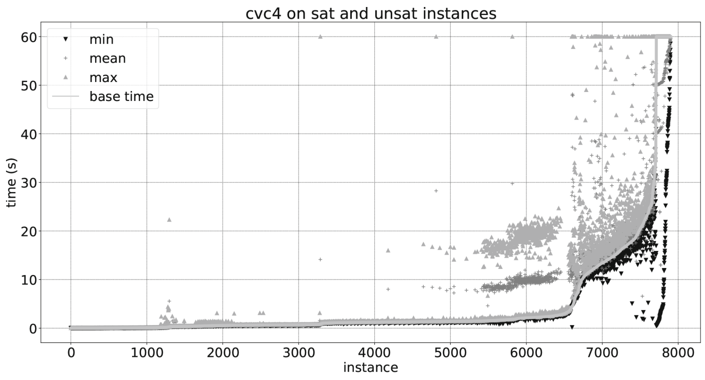
The cycle now repeats with cvc5 (version 1.0.0), and the same question comes up again. The question we are evaluating yet again is, "do we upgrade the existing cvc5 version with the latest or add yet another version of CVC to the portfolio solver". Some early experiments show that there is no clear answer yet. The results so far comparing the different version of cvc5 shown in Fig. 5(e) and (f) indicate that the latest version of cvc5 is not monotically better in performance than either of its previous versions. We do want to leverage the better proof generating capabilities of cvc5 (version 1.0.0) in order to gain more assurance in the correctness of the UNSAT queries.

4.2 Stability of the Solvers

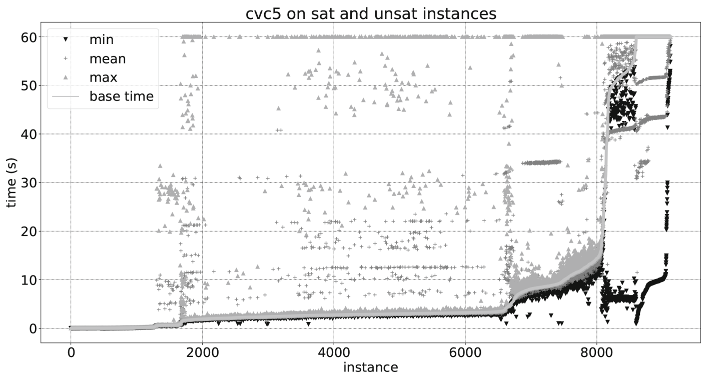
We have spent quite a bit of time defining and implementing the encoding of the AWS access control policies into SMT. We update the encoding as we expand to more use cases or when we support new features in AWS. This is a slow and careful process that requires expertise in understanding AWS and how SMT solvers work. There is a lot of trial and error to figure out what encoding is correct and performant.

To illustrate the importance of the encoding, we present an experiment on solver runtimes with different ordering of clauses for our encoding (Fig. 6). For the same set of problem instances used in Fig. 5, we now use the standard SMT competition shuffler[2] to reorder assertions, terms, and rename variables to study the effect of ordering clauses for our default encoding. In Fig. 6, each point on the x axis corresponds to a single problem instance. For the problem instance, we run it in its original form (default encoding) which is the "base time", and

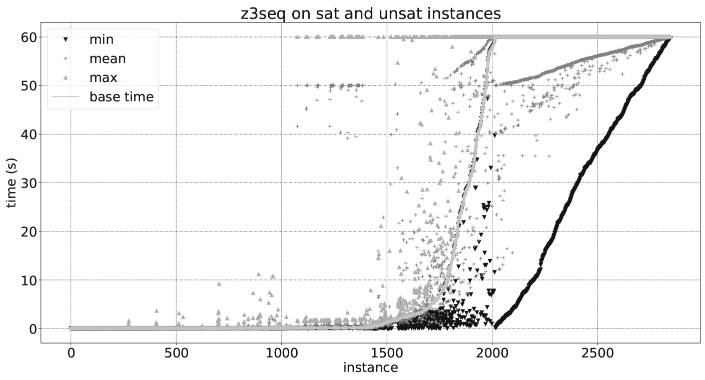
[2] https://github.com/SMT-COMP/scrambler.



(a) CVC4



(b) cvc5 version 0.0.7



(c) Z3 sequence string solver

Fig. 6. Variance in runtimes after shuffling terms in the problem instances.

five shuffled versions. This gives us a total of six versions of the problem; we record the min, max, and mean times. So for each problem instance, x we have:

1. (x, base time): time on the original problem;
2. (x, min time): minimal time on the original and 5 shuffled problems;
3. (x, max time): maximal time on the original and 5 shuffled problems; and
4. (x, mean time): mean time on the original and 5 shuffled problems.

The instances are sorted by 'base time' so the line looks smooth in base time, and the other points look more scattered. The comparison between CVC4 in Fig. 6(a) and Fig. 6(b) cvc5 shows that cvc5 can solve more problems with the default encoding shown by the smooth base line. However, when we shuffle the assertions, terms and other constructs in the problem instance, the performance of cvc5 varies more dramatically compared to that of CVC4. The points for the maximal time are spread wider across the graph and there are now several time-outs in Fig. 6(b).

4.3 Concluding Remarks

Based on our experience from generating a billion SMT queries a day, we propose some general areas of research for the community. We believe these are key to enabling the use of solvers to evaluate security controls, and to enable applications in emerging technologies such as quantum computing, blockchains, and bio-engineering.

Monotonicity and Stability in Runtimes. One of the main challenges we encountered is the lack of monotonicity and stability in runtimes within a given solver version and across different versions. Providing this stability is a fundamentally hard problem due to the inherent randomness in SMT solver heuristics, search strategies, and configuration flags. One approach would be to incorporate the algorithm portfolio approach [31,34,42] within mainstream SMT solvers. A way enable algorithm portfolio is to leverage serverless and cloud computing environment, and develop parallel SMT solving and distributed search strategies. At AWS, this is an area that we are investing in as well. There has been some work in parallel and distributed SMT solving [41,45] but we need more. Another aspect of research would be to develop specialized solvers that focus on a specific class of problems. The SMT-comp could devise categories that allow room for specific types of problem instances as an incentive for developing these solvers.

Reduce the Barrier to Entry. Generating a billion SMT queries day is a result of the exceptional work and innovation of the entire SMT community over the past 20 years. A question we are thinking about is how to replicate the success described here for other domains in Amazon and elsewhere. There is a natural tendency in the formal methods community to target tools for the expert user. This limits their broader use and applicability. If we can find ways to lower the barrier to adoption, we can gain greater traction and improve the security, correctness, availability, and robustness of more systems.

More Abstractions. SMT solvers are powerful engines. One potential research direction for the broader community is to provide one or more higher level languages that allows people to specify their problems. We could create different languages based on the domain and take into account the expectations of developers. This would make interacting with a solver a more black-box exercise. The success we have had with SMT in Amazon, can be recreated in other domains if we provide developers the ability to easily encode their problems in a higher level language and use SMT solvers to solve them. It will more easily scale by not requiring a formal methods expert as an intermediary. Developing new abstractions or intermediate representations could be one approach to unlock billions of other SMT queries.

Proof Generation. All SMT solvers should be generating proofs to help the end-user gain confidence in the results. There has been some initial work in this area [9,20,27,43,44],but SMT has a long way to catch up with SAT solvers, and for good reason. The proof production is important for us gain greater confidence in the correctness of our answers, though it creates a tension with the timeliness. We need the proof production to be performant and the tools that check the generated proofs to be correct themselves. Continued push on different testing approaches, including fuzzing and property-based testing of SMT solvers, should continue with the same rigor and enthusiasm. Using these fuzz testing and mutation testing based techniques in the development workflow of SMT solvers is something that should become mainstream.

We are working to provide a set of benchmarks that can be leveraged by SMT developers to help further their work, are funding research grants in these areas, and are willing to evaluate new solvers.

References

1. Encryption SDK Dafny model. https://github.com/aws/aws-encryption-sdk-dafny
2. s2n bignum verification. https://github.com/awslabs/s2n-bignum
3. Almeida, J.B., et al.: A machine-checked proof of security for AWS key management service. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 63–78 (2019)
4. Athanasiou, K., Cook, B., Emmi, M., MacCarthaigh, C., Schwartz-Narbonne, D., Tasiran, S.: SideTrail: verifying time-balancing of cryptosystems. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 215–228. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_12
5. Backes, J., et al.: Reachability analysis for AWS-based networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 231–241. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_14
6. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–9. IEEE (2018)
7. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM **54**(7), 68–76 (2011)

8. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pp. 203–213 (2001)
9. Barbosa, H.: New techniques for instantiation and proof production in SMT solving. Ph.D. thesis, Université de Lorraine (2017)
10. Barbosa, H., et al.: cvc5: Versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022. Lecture Notes in Computer Science, vol. 13243. LNCS, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
11. Barrett, C., et al.: cvc5 at the SMT competition 2021
12. Barthe, G., et al.: EasyCrypt: a tutorial. In: Aldini, A., Lopez, J., Martinelli, F. (eds.) FOSAD 2012-2013. LNCS, vol. 8604, pp. 146–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10082-1_6
13. Bayless, S., et al.: Debugging network reachability with blocked paths. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 851–862. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_39
14. Bayless, S., Bayless, N., Hoos, H., Hu, A.: Sat modulo monotonic theories. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 29 (2015)
15. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press (2009)
16. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. SMT **12**, 76–86 (2012)
17. Bodden, E.: Inter-procedural data-flow analysis with IFDS/IDE and soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, pp. 3–8 (2012)
18. Bornholt, J., et al.: Using lightweight formal methods to validate a key-value storage node in Amazon s3. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp. 836–850 (2021)
19. Bouchet, M., et al.: Block public access: trust safety verification of access control policies. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 281–291 (2020)
20. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 151–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
21. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering, pp. 73–82 (2004)
22. Chong, N., et al.: Code-level model checking in the software development workflow. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 11–20. IEEE (2020)
23. Chudnov, A., et al.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_26
24. Cook, B.: Formal reasoning about the security of Amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3

25. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 467–486. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_28
26. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. Formal Methods Syst. Des. **57**(1), 34–52 (2021)
27. Deharbe, D., Fontaine, P., Paleo, B.W.: Quantifier inference rules for SMT proofs. In: First International Workshop on Proof eXchange for Theorem Proving-PxTP 2011 (2011)
28. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.P.: Safe asynchronous event-driven programming. ACM SIGPLAN Notices **48**(6), 321–332 (2013)
29. Emmi, M., et al.: Rapid: checking API usage for the cloud in the cloud. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1416–1426 (2021)
30. Gastin, P., Moro, P., Zeitoun, M.: Minimization of counterexamples in SPIN. In: Graf, S., Mounier, L. (eds.) SPIN 2004. Minimization of counterexamples in spin, vol. 2989, pp. 92–108. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24732-6_7
31. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1–2), 43–62 (2001)
32. Groce, A., Kroening, D.: Making the most of BMC counterexamples. Electron. Notes Theoret. Comput. Sci. **119**(2), 67–81 (2005)
33. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
34. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(5296), 51–54 (1997)
35. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
36. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 141–153. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_15
37. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
38. Lewis, J.R., Martin, B.: Cryptol: High assurance, retargetable crypto development and validation. In: IEEE Military Communications Conference, 2003. MILCOM 2003. vol. 2, pp. 820–825. IEEE (2003)
39. Liang, T., et al.: An efficient SMT solver for string constraints. Formal Methods Syst. Des. **48**(3), 206–234 (2016)
40. Luo, L., Schäf, M., Sanchez, D., Bodden, E.: Ide support for cloud-based static analyses. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1178–1189 (2021)
41. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 428–443. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_27

42. Rice, J.R.: The algorithm selection problem. In: Advances in Computers, vol. 15, pp. 65–118. Elsevier (1976)
43. Stump, A., Oe, D.: Towards an SMT proof format. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, pp. 27–32 (2008)
44. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods Syst. Des. **42**(1), 91–118 (2013)
45. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_60
46. Zheng, Y., Zhang, X., Ganesh, V.: Z3-STR: a z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 114–124 (2013)